Self-Maintaining Overlay Data Structures for Pervasive Autonomic Services

Marco Mamei and Franco Zambonelli

Dipartimento di Scienze e Metodi dell'Ingegneria,
University of Modena and Reggio Emilia
Via Allegri 13, 42100 Reggio Emilia, Italy
{mamei.marco, franco.zambonelli}@unimore.it

Abstract. Overlay data structures are a powerful mechanism to provide application components with context-information and to let them interact in dynamic-network scenarios like mobile ad-hoc networks and pervasive computing. These overlays can be propagated across a network in order to support components' context awareness and coordination activities. We present a modeling framework and some autonomic algorithms to create overlay data structures that are able to self-maintain their intended distribution under a number of circumstances. The paper presents some experiments and performance measures to validate our approach and to show its scalability.

1 Introduction

One of the main challenges for developing distributed applications in pervasive computing scenarios is to provide application components with autonomic and adaptive coordination mechanisms capable of sustaining the dynamics of the operational environment and of adapting to different contexts.

A number of recent researches try to address this problem by relying on overlay data structures. Overlay data structures are distributed data structures encoding specific aspects of the application components' operational environment. These overlays are propagated across a network by a component in order to represent and "communicate" its own activities. Overlay data structures are easily accessible by the components and provide easy-to-use context information (i.e., the overlays are specifically conceived to support their access and fruition).

The strength of these overlay data structures is that they can be accessed piecewise as the application components visit different places of the distributed environment. This lets the components to access the right information at the right location. In addition, overlay data structures decouple components' activities from the underlying network dynamism. Components interacting and perceiving their operational environment by means of these overlay data structure can disregard the underlying physical network and its dynamics.

From this point of view, overlay data structures enable "stigmergy" [1] in that components' interactions can be mediated by these kind of overlay "markers" distributed across the environment. From another perspective, overlay data

structures generalize the idea of overlay networks. Overlay networks are basically routing distributed data structures providing components with a suitable application-specific view of the network (i.e. they allow components to perceive a specific overlay topology of the network) [2]. Overlay data structures do not focus on network topology only. They are general-purpose and can encode any kind of context information.

To clarify these concepts let us focus on the problem of coordinating the movements of some autonomous components (i.e., agents) in a distributed environment [3]. Hereafter we will use the term agent to refer to any autonomous real-world or software entity with computing and networking capability (e.g., a user carrying on a Wi-Fi PDA, a robot, or a modern car). In particular, we focus on the simple application of having two persons, provided with a PDA, moving across an environment instrumented with an ad-hoc network infrastructure. The goal of the application is to allow one person to be guided by the PDA, to follow the other person. A simple solution based on overlay data structures is the let the person to-be-followed to spread in the environment (i.e., ad-hoc network) a data structure that increases an integer value by one at every hop as it gets farther from the source. This creates a sort of gradient that can be followed downhill by the other person to complete the application [4] (see Figure 1(a)). If the person to-be-followed moves, it is important that the overlay data structure adjust its shape accordingly, so that the gradient leads to that person anyway (see Figure 1(b)). The power of this approach is that the overlay data structure provides expressive contextual information tailored for that specific task. The agent running on the PDA does not need to know any map of the environment, nor it has to execute complex algorithms to decide where to go. It just blindly follows the overlay data structure.

Beside this exemplary application, overlay data structures are general purpose and can be applied in a wide range of application scenarios, ranging from robotics to network routing (see the following section for a brief review of their use in this context).

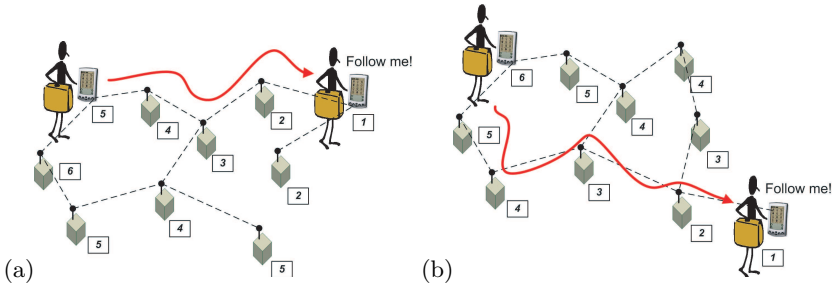


Fig. 1. (a) A gradient overlay data structure enables an agent to follow another one. (b) The data structure is updated to reflect the new agent position.

The main contribution of this paper is to present a general framework to model and implement overlay data structures in dynamic network environments. In particular, we focus on some decentralized algorithms that enable overlay data structures to maintain their intended distribution despite network topology changes.

A number of experiments is reported to assess the efficiency of our approach. In particular, the experiments show that the proposed algorithms are scalable in that the operations to maintain an overlay data structure tend to be confined near the place where the data structure had been corrupted. Moreover, some experiments to evaluate the time taken to propagate, delete and maintain such overlay data structures have been conducted. These experiments show that the time required to fix an overlay is a specific (bell-shaped) function of the distance from where the data structure get corrupted. This time drops almost to 0 when the distance is big enough confirming our scalability results.

2 Application Scenarios and Related Work

In this section, we first introduce several application scenarios where such overlay data structures have been successfully applied. The contribution of this brief survey is to illustrate the wide range of scenarios that benefit from these overlay data structure and to show their generality. Then, we present currently proposed approaches that can be employed to create and maintain these data structures, showing their shortcomings. Both these aspects are the main motivations at the root of our proposal that will be described later.

2.1 Application Scenarios

Motion Coordination. As already stated in the introduction, overlay data structures, spread across a properly networked environment, have been used in [3, 4] for the sake of enabling agents (e.g. users carrying a PDA, robots, cars) to coordinate their respective movements. The goals of agents' coordinated movements can be various: letting them to meet somewhere, distribute themselves accordingly to specific spatial patterns, or simply move in the environment without interfering with each other and avoiding the emergence of traffic jams. As previously stated, overlay data structures provide suitable tools for this task, in fact, they can be accessed piecewise to guide agents motion step-by-step.

Routing in Mobile Ad-Hoc Network. Routing can be easily modelled as a coordination problem: agents (i.e. network nodes) need to cooperate forwarding each other messages to enable long-range, multi-hop communication. The main principle underlying many routing algorithms is to build several overlay data structures (implemented by means of a set of distributed routing tables) suitable to provide route information. Specifically, these data structures create paths in the network enabling agents to forward messages in the right direction. These paths (i.e. data structures) are maintained to take into account changes in the network topology [5]. The idea at the basis of distributed routing data

structure is the same as motion coordination: provide agents with a ready-to-use representation of the context (i.e. where the message should go next).

Swarm Intelligence. From a general perspective, overlay data structures are at the core of a number of swarm-intelligent (e.g. ant-inspired) systems [1]. These approaches mimic the way in which social insects, like ants, coordinate their activities to achieve complex tasks (e.g. the mechanism used by ants to find food can be used in the context of computer networks to route packets or find relevant resources [1, 6]). The key to these approaches is in emulating the way in which ants interact one another. They do so by means of pheromone signals they spread in the environment that will be perceived by other ants later on. These pheromone signals can be used to find food sources, or to coordinate efforts in moving some heavy objects, etc. Pheromone signals can be easily modelled by means of overlay data structures. Overlay data structures implementing the concept of pheromone could be distributed by the agents themselves as they move across the network. These data structures can then be used as trails driving agents' activities. The research projects Anthill [6] and SwarmLinda [7] share the idea of applying ant-inspired algorithms to Internet-scale Peer-to-Peer systems. Here, overlay data structures - modelling ants' pheromones - create paths connecting peers that share similar files, thus enabling, for example, an effective content-based navigation in the network of peers.

Amorphous computer. Overlay data structures are at the core of the amorphous computer [8] research. An amorphous computer consists of massive numbers of identically-programmed and locally-interacting computing agents, embedded in space. It can be modelled as a collection of "computational particles" sprinkled randomly on a surface or mixed throughout a volume. Overlay data structures can be spread and deployed in the amorphous computer to let various patterns and shapes emerge among the computational particles. Just to mention few trivial examples, if a leader particle spreads a hop-increasing overlay data structure (as defined above), it is possible to create approximately circular regions of controlled size: particles sensing the overlay are able to determine if they are in or out a specific circular region of radius R (i.e. they are in if they sense the data structure with a value lower than R). Similarly, if a line of particles propagate the above data structure, stripes instead of circles can be identified in the amorphous computer [8].

Modular Robotics. A modular or self-reconfigurable robot is a collection of simple autonomous actuators with few degrees of freedom connected with each other. A distributed control algorithm is executed by all the actuators to let the robot assume a global coherent shape or a global coherent motion pattern (i.e. gait). Some proposed approaches adopt overlay data structures to control such a robot [9]. A distributed shape or motion gait is encoded by means of overlay data structures spread across the robot specifying how the robot's actuators should bend. Robots are programmed to bend their actuators depending on the sensed data, thus realizing the prescribed motion gait.

2.2 Related Work

In all the above examples it is clear that the self-maintained property for the data structures being used in an asset of tremendous importance. In fact, failures in the data structures typically break down the application. For example, in a motion coordination task, broken overlay data structures are perceived as dead-ends by the application agents (i.e., agents follow an overlay that does not lead where it is supposed to). As another example, in modular robotics, broken overlay – providing wrong information on where to bend the actuators – can disrupt the the global gait. On the contrary, if the overlays self-adapt their structure to remain coherent, the robot can keep going despite module failures and even partitions (i.e., a robot that split in two "produces" two smaller robots that are still able move).

Given the wide range of application scenarios and the importance of maintaining the overlay data structures, it is not surprising that some mechanisms to create and maintain such data structures have already been proposed. In very general terms, there are two main approaches proposed in literature: proactive algorithms [10, 11] and reactive algorithms [5].

Proactive algorithms keep-up the overlay data structures by simply letting agents to repropagate them on a time-basis. If repropagations are more frequent than network changes, the data structures maintain (almost always) their intended distribution. Although some mechanisms (e.g., in sensor network [11]) propose to aggregate overlay data structures to save bandwidth [10], the problem of this approach is about its scalability. Large and dynamic networks are likely to be saturated by repropagations. A partial relief from this problem comes from opportunistic and epidemic algorithms [12, 13]. These are proactive algorithms that, instead of flooding the network to update the whole structure, update only sub-parts of the overlay.

Reactive algorithms, originally introduced as mobile ad-hoc network routing algorithms, propose not to maintain the overlay data structures. Network dynamics can break the overlay data structures, but, when a node need to access one of them, it engages a discovery protocol to find the correct value of the data structures [5, 14]. Basically, such protocol amounts at finding - via a flood-based mechanism - the source of the overlay data structure, then to let the source resend the correct data accordingly to the new network conditions. This approach is very effective if nodes are highly mobile and they have to communicate only sporadically. On the contrary, if nodes need to constantly access the overlay data structures - like in most of the above mentioned applications - this approach can trash performance.

The main contribution of this paper is to try to overcome such proposals and to present some algorithms trying to combine the best of both proactive and reactive approaches to let the overlay data structures self-maintain, without flooding the network while still continuously providing components with the correct data structures' values.

3 A Framework to Model Overlay Data Structures

In this section we present a modeling framework to create self-maintained overlay data structures. It is important to remark that these overlays are at the core of the distributed middleware infrastructure called Tuples On The Air (TOTA) developed within our group [3, 4]. However, in this paper we do not discuss such a middleware and focus only on the data structures themselves: how they can be created and which algorithms are needed to support them.

Overlay data structures can be defined by means of a couple *(C,P)*.

The content C can be an arbitrary data structure representing the information carried on by the data structure. The propagation rule P determines how the overlay data structure should be distributed and propagated across the network. This includes determining the "scope" of the overlay (i.e. the distance at which it should be propagated and possibly the spatial direction of propagation) and how such propagation can be affected by the presence or the absence of other data structures in the system. In addition, the propagation rules can determine how the content C should change while it is propagated. Overlay data structure are not necessarily distributed replicas: by assuming different values in different nodes, they can be effectively used to shape a structure expressing some kind of contextual and spatial information. In addition, a maintenance mechanism should be enforced to let the overlay data structure preserve its intended distribution *(C,P)* despite network contingencies.

The idea of overlay data structures can potentially be implemented on any distributed middleware providing basic support for data storing (to store data values), communication mechanisms (to propagate overlay data structures) and event-notification mechanisms (to update overlay data structures and notify agents about changes in overlay data structures' values).

In most of the application scenarios, described in section 2.1, there are two main kind of overlay data structures being employed. They will be described in the next subsections.

3.1 Hop-based Overlay Data Structures

Hop-based overlay data structures are those having a content (C) and a propagation-rule (P) that depend only on the hop-distance from the source.

Hop-based overlay data structures enable to express and diffuse across the network (possibly within a bounded scope) contextual information related to the network distance from the source. These kind of data structures have been widely used in motion coordination, routing and location-based-information-access applications [4, 8].

We designed the *Hop* data structure as a basic template to build this kind of overlays. In particular, the *Hop* data structure has a integer hop-counter (hop) as a content. Once one of these data structures is injected in the network, it propagates breadth-first maintaining the hop-distance from the source. It is clear that, once the above hop-distance is maintained within the data structure, adding other parameters or triggering conditions on the basis of such hop-value

becomes trivial (e.g., add the parameter k = f(hop) , or the condition "stop propagating if hop > 5").

These overlay data structures have to be maintained despite network topology changes either due nodes mobility or failures. The self-maintaining algorithm that will be described next performs exactly this task. The strength of these data structures, from a software engineering point of view, is that agents have simply to inject these data structures without further taking care of their update. All the burden in maintaining data structures is moved away form the agents.

3.2 Space-based Overlay Data Structures

Space-based overlay data structures are those having a content (C) and a propagation-rule (P) that depend on the spatial coordinates (i.e. location) of the node hosting the data. To realize this kind of overlay data-structures it is thus fundamental to provide agents with a suitable localization mechanism. Localization can be either realized by means of specific hardware devices or also via specific self-localization algorithms, discussed later, that use hop-based overlay data structures. These overlays enable to express contextual information related to the spatial location of the application agents. They have been widely used in sensor network scenarios and peer-to-peer application. [15, 4].

We designed two main types of space-based overlay data structures: *Metric* and *Space* data structures. They have three numbers (x,y,z) as a content. Once one of these data structures is injected in the network, it propagates changing its content so that (x,y,z) reflect the coordinates of the node in a coordinate system centered where the data structures was first injected. It is clear that, once the above coordinates are maintained within the data structure adding other parameters, or triggering conditions on the basis if such coordinates is trivial. We will describe the differences between these data structures (related to their maintenance algorithm) in the next section.

The design of *Metric* and *Space* data structures, given the availability of a GPS-like device, is straightforward. Once injected, the data structures will store the injecting node GPS coordinates and will initialize its content to (0,0,0). Upon reaching a new node, it will change the content to the GPS coordinates of the new node translated back by the injecting node coordinates.

It is also very interesting to report that some recently proposed localization algorithms (e.g. beacon-based triangulation) [16] relies on *Hop* overlay data structures to create a coordinate system over the network. Basically, in these algorithms a number of beacon-nodes spread across the network *Hop* data structures expressing the hop-distance from themselves together with their coordinates in some (possibly arbitrary) reference frame. Other nodes infer their own coordinates by triangulating the distances from these beacon nodes. Such coordinate system can then be used as the basis for space-based overlay data structures (they act as a GPS-like device). Self-maintenance of hop-based overlays assures that the coordinate system remains always up to date. Self-maintenance of the space-based overlays enable such data structures to be consistent despite low-level hop-based adjustments.

4 Self-Maintenance Algorithms

In this section we present the main contribution of this paper. It consists of two algorithms to enable hop-based and space-based overlay data structures to self-maintain their distribution despite network dynamism. For the sake of clarity, in the followings, we will use the term *data structure* to refer to the whole distributed overlay. We will use the term *data* to indicate a single piece of the overlay stored in a single node. For example a *Hop data structure* is made of a number of *data* each stored in a node of the network.

4.1 The Hop-Based Self-Maintenance Algorithm

The most significant algorithm we will describe is the one allowing *Hop* data structures to self-maintain their shape despite network dynamism. For obvious scalability reasons, we would like the burden of such algorithm to be evenly distributed among all the network nodes. Recall that a *Hop* data structures propagates increasing its integer content by one at every hop. Given a local instance of such a data X, we will call another data Y a *supporting data* of X if: Y belongs to the same distributed data structure as X, Y is one-hop distant from X, the value of Y is equal to the value of X minus one. With such a definition, a supporting data of X is a data that could have created X during its propagation. Moreover, we will say that X is in a *safe-state* if it has at least a supporting data, or if it is in the node that first injected the data structure (i.e. hop value = 0). We will say that a data is not in a safe-state if the above condition does not apply (i.e. it has not any supporting data and it has a hop value greater that 0).

The basic idea is that a data that is not in a safe-state should not be there, since no neighbor data could have created it.

Each local data can subscribe to the arrival or the removal of other data of its type in its one-hop neighborhood. Upon a removal, each data reacts by checking if it is still in a safe-state. In the case a data is not in a safe state, it erases itself from the local node. This eventually causes a cascading deletion of data until a safe-state data can be found, or the source is eventually reached, or all the data in that connected sub-network are deleted.

In the case a data is in a safe-state, the removal of neighbor data triggers a reaction in which the data propagates to that node. It is worth noting that this mechanism is the same as when a new node is connected to the network. Similar considerations apply with regard to data arrival: when a data senses the arrival of a data having a value higher than its own plus one, it means that, because of topology changes, a short-cut leading to the source has been created. In such a situation the data can propagate to the new node to overwrite the previous data, fixing the data structures shape.

This set of mechanism is enough to make *Hop* data structures self-maintain.

To prove the validity of this algorithm we will show its correctness with regard to four special cases (see figure 2). The rationale behind these four spacial cases is answering the following questions: does the network topology change implies a link creation or removal? Is the changed link the only one connecting two

networks or there are others? It is rather clear that the four possible yes/no answers to these questions (four special cases) can be generalized to cover all the other possibilities (i.e, all the possible topology reconfigurations). The four special cases proving the correctness of *Hop* data structures self-maintenance are reported in figure 2. In figure 2(a) the link between A and B breaks down. Since the data on B has not any supporting data, it is not in a safe state anymore. Thus it deletes itself. After that, the data on C does not see any supporting data, thus it deletes itself. This applies recursively in all the bottom network. After that, the distributed data structure is in a consistent state with respect to the new topology. In figure 2(b) a new link between A and B is created. The data on A propagates to B and then recursively to the all bottom network. In figure 2(c) the link between A and B breaks down. Like in case (a) this causes a cascading deletion until reaching a safe-state data. In this example, the safe-state data is in node D. When this latter data sees that the data on C gets deleted it can propagate toward C fixing the gap. The propagation applies recursively to the whole bottom network, adjusting the distributed data structures. In figure 2(d) a new link is created between C and D. The data on D finds in its neighbor a data with value greater than its own plus one (i.e. $n+2 > k+1$). Thus, it propagates to C, overwriting the data on C. This process applies recursively until the right node is found (i.e. where the two branches of the data structure seamless merge: $n+1 = k+2$).

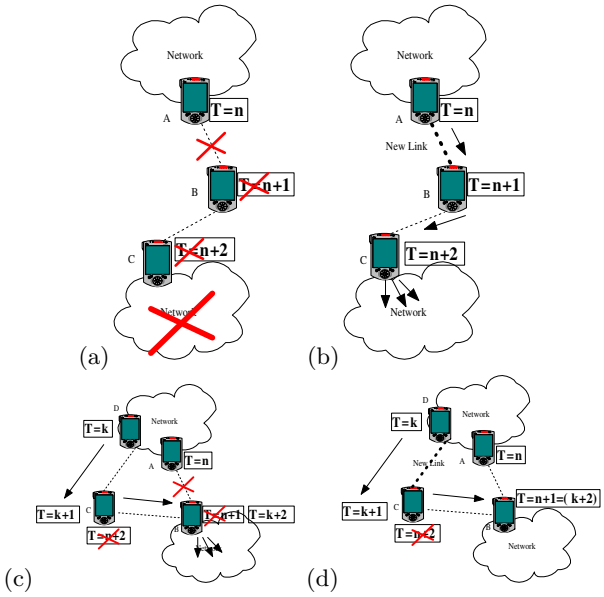


Fig. 2. Hop-maintenance in four special cases

Clearly if the network constantly changes its topology faster than the time required to the algorithm to fix the data structure shape the process could never converge. However the self-maintenance algorithm ensures that, once the network stabilizes, the algorithm eventually makes the overlay distributed data structure converge to a consistent state.

4.2 The Space-Based Self-Maintenance Algorithm

Given a reliable coordinate system, maintaining space-based overlay data structures is rather easy. Once these data structures have been propagated, if a node moves (not the source one), only the data local value is affected, while all the others are left unchanged. In fact, the others physical positions with respect to the source do not change. In particular, when a node moves (not the source one), the data structure locally changes its content by accessing the new GPS information.

To understand the difference between *Metric* and *Space* data structures, it is fundamental to focus on what happens when the source moves. In theory, all the data of the overlay must be changed because the origin of the coordinate system has shifted. This is exactly what happens in *Metric* data structures where the origin of the coordinated system is anchored to the source node. This of course can lead to scalability problems, especially if the source is highly mobile. What happens if also the source updates its value locally, without further propagating? In this case, the origin of the coordinate system remains where the data structure was first injected, even if no nodes are in that position. The coordinate system is maintained by the network, but not affected by it. This is the *Space* data structure implementation.

5 Performance and Experiments

The effectiveness of our approach is of course related to costs and performance in managing overlay distributed data structures.

5.1 Overhead

The cost of propagating a data structure, relying on a multi-hop mechanism, is something inherently scalable. Each node will have to propagate the data structure only to its immediate neighbors. The size of the network does not matter since the global effort to spread the data structure is fairly partitioned between the constituting nodes.

The scalability of data structures maintenance is less clear. The main requirement for our algorithms is to be independent of the network size. This implies maintenance operations must be confined within a locality from where events that altered the data structure (e.g., a network topology change) happened. If it is so, concurrent events happening at distant points of the network do not accumulate locally. If on the contrary maintenance operations always spread across

the whole network, distant concurrent events do accumulate and the system does not scale.

With regard to *Hop* data structures, establishing if maintenance operations are confined to an area neighboring the place in which the network topology had actually changed is rather complicated. The size of this neighborhood is not fixed and cannot be predicted a-priori, since it depends on the network topology.

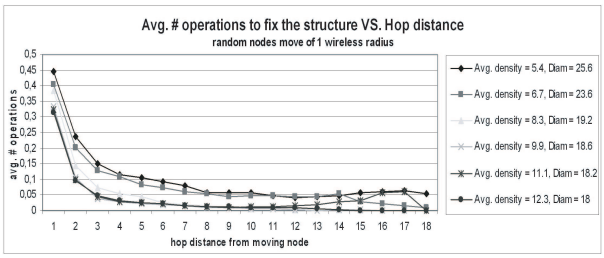
Thus, trying to answer, we exploited a network simulator developed within our group [4], performing a large number of experiments to measure the scope of maintenance operations. To perform the experiments, we run several simulations varying the node density and their initial position. In particular, we run six sets of experiments where we randomly deployed 200, 250, 300, 350, 400, 450 nodes in the same area; thus obtaining an increasing node density and a shrinking network diameter. All the experiments were repeated a large number (over 100) of times with different initial network topologies and the result were averaged together. The experiment consisted in a randomly chosen node injecting a *Hop* data structure in the network. After that, randomly chosen nodes start moving independently (following a random waypoint motion pattern) perturbing the network. In particular, a randomly picked node moves randomly for a distance equals to 1 wireless radius. This movement changes the network topology by creating and disrupting links. The number of messages sent between nodes to adjust the data structure, according to the new topology, is recorded. Specifically, we evaluate the average number of messages exchanged by nodes located at x-hop away from the moving node. Then, we average these numbers over a large set of topology changes. The results of this experiment are in figure 3(a).

The experiments reported in figure 3(b) have been conducted in the same manner. This time, however, nodes move for a distance of 1/4 wireless radius. This second set of experiments is intended to show what happens for very little topology reconfigurations (wider reconfigurations can be depicted as a chain of these smaller ones).

The most important consideration we can make looking at the figure is that, when a node moves and the network topology changes consequently, a lot of update operations will be required near the area where the topology changes, while only few operations will be required far away from it. This implies that, even if the network and the data structures being propagated have no artificial boundaries, the operations to keep their shape consistent are strictly confined within a locality scope. This result is even more significant if compared to the average network diameter (averaged over the various experiments). It is easy, in fact, to see that the number of operations required to maintain a data structure falls close to zero well before the average diameter of the network, thus confirming the quality of our results. This fact supports the idea that the operations to fix distant concurrent topology changes do not add up, making the system scalable.

With regard of Metric and Space Data data structures, determining if their self-maintenance operations are confined is rather easy. In section 4.2 we described that *Metric* data structures's maintenance operations are confined to the only node that moved for all the nodes apart from the source, while it

(a)



(b)

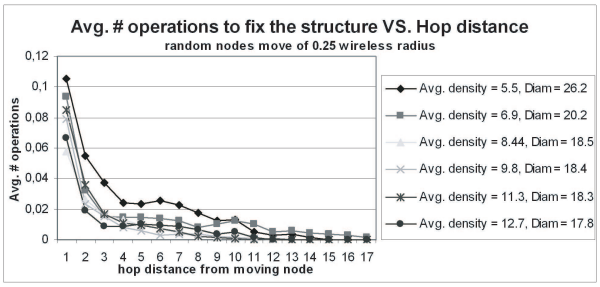


Fig. 3. The number of maintenance operation decreases sharply with the hop distance from topology reconfigurations caused by: (a) random node movements for 1 wireless radius. (b) random node movements for 1/4 wireless radius.

spreads across the whole network if the source moves. So *Metric* data structures must be used carefully, maybe with custom rules in their propagation rules limiting a-priori their scope, or by triggering update operations only if the source node moves by at least a certain amount (e.g. trigger update only if the source moves for at least 1m). The answer for a *Space* data structures, instead, is clearly affirmative since maintenance is strictly locally confined.

How much time is required to spread a data structure across the network? How much time to delete the data structure? How much time to let a data structure maintain its shape? These are fundamental questions to evaluate our algorithms.

5.2 Timing

The time to propagate or to delete a data structure can be easily computed in theory. Let us focus on the basic operation of a *Hop* overlay structure travelling from a node to another neighbor node and let us assume the average time to perform such operation as the unity $T_u = 1$ of our timing model. More in detail, we can define T_u as: $T_u = T_{prop} + T_{send} + T_{travel} + T_{rcv}$. T_{prop} is the time taken by a data structure to run its algorithm on a node. T_{send} is the time required to serialize and send the data content. T_{travel} is the time to letting the stream of data arrive on the other node. T_{rcv} is the time to receive, deserialize the data and have it ready again to execute its methods. This abstract timing model allows us

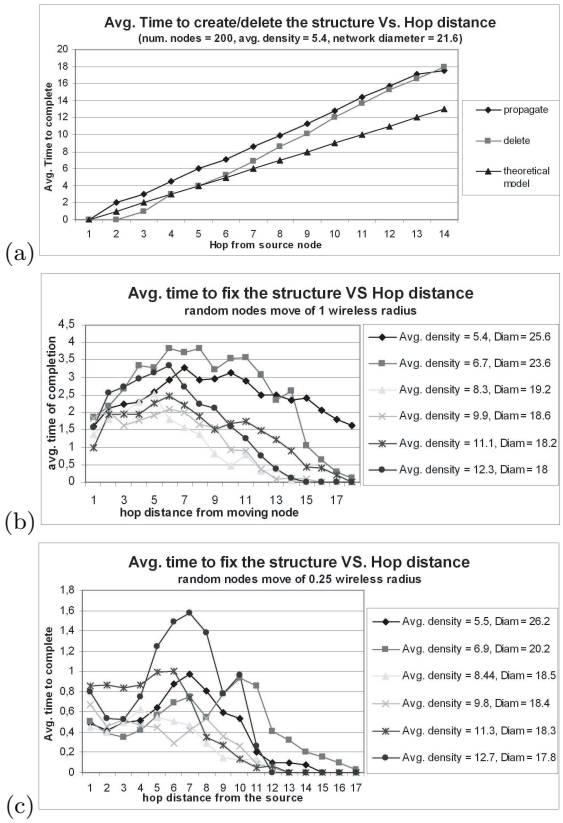


Fig. 4. (a) Time required to propagate and delete a data structure: theoretical model and simulations experiments. (b) The time required to fix a data structure randomly selected nodes move of 1 wireless radius. (c) Nodes move of 1/4 wireless radius.

to abstract from low-level details such as the network being used and the time to parse and process the data structures. All this details are wrapped by our abstract unity.

Under this hypothesis, it is rather easy to understand that a data structure will propagate at X hop distance in a X time. To assess this theoretical model, we performed several experiments with our simulator. In particular, a randomly selected node injects several overlay data structures in the network. The time taken by the data structures to reach a distance of x-hop away from the source has been recorded. The results were averaged together over a large set of experiments. The results are depicted in figure 4(a). Looking at the graph, we notice a small disagreement between the theoretical model and experimental results. This can be easily explained, considering that data structures propagation does not happen always in a perfect expanding ring manner. To correct the

imbalances and consequent backward propagations, some extra operations are required. These extra operation account for the time gap between the theoretical model and experimental results.

Evaluating in theory the time to fix a *Hop* data structure is not easy since it depends on the topology of the network. Given this problem, we focused again on simulations to test the system performance, and we considered the same experimental set up of overhead experiment. In this set of experiments, however, instead of counting the number of operations required to fix the structure, we measured the time taken. In particular, for a given network reconfiguration (caused by moving nodes), we recorded the time at which a node performs the last operation to fix the overlay data structure. These times are grouped by the hop distance from the moving node and averaged together. These operations have been repeated several (more then 100) times and all the outcomes have been averaged together to obtain the results depicted in figures 4(b-c).

In these experiments it is possible to see that the time to complete maintenance operations has a rough bell-shaped behavior. It has a low value near the topology problem. This is because, the data close the the topology change is the first to be maintained (so they complete maintenance in short time). Then it increases with the hop-distance, since it requires time to the local algorithm to propagate information across the network to delete and update those data that must be maintained. Even further, it decreases because of the fact that maintenance tend to remain confined, and so a lot of data"maintain" in 0 time.

With regard to the other overlay data structures; *Space* data structures are maintained with only one-hop-bounded operations, so they always maintain with a delay equals to 1. *Metric* data structures either maintain with one-hop-bounded operations, and so with a delay of 1, or if the source moves, they are repropagated and thus timing evaluation falls in the propagation and deletion case (previous subsection).

6 Conclusion and Future Work

In this paper we have presented a modeling framework and some algorithms to create self-maintained overlay data structures. Such overlay data structures have been proven useful in developing a number of application tasks in the context of distributed computing. Our future work will be mainly devoted in researching on how the presented overlay data structures can be extended and generalized to cover more diverse application tasks.

7 Acknowledgments

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

References

1. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence. From Natural to Artificial Systems. Oxford University Press, Oxford, United Kingdom (1999)
2. Ratsanamy, S., Francis, P., Handley, M., Karp, R.: A scalable content-addressable network. In: ACM SIGCOMM Conference. ACM Press, San Diego, CA, USA (2001)
3. M. Mamei, F.: Field-based Coordination for Pervasive Multiagent Systems. Springer Verlag (2006)
4. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the tota middleware. In: IEEE International Conference On Pervasive Computing. IEEE CS Press, Orlando, FL, USA (2004)
5. Butera, W.: Embedded networks: Pervasive, low-power, wireless connectivity (2001) PhD Thesis, Massachusstes Institute of Technology.
6. Babaoglu, O., Meling, H., Montresor, A.: A framework for the development of agent-based peer-to-peer systems. In: International Conference on Distributed Computing Systems. IEEE CS Press, Wien, Austria (2002)
7. Menezes, R., Tolksdorf, R.: A new approach to scalable linda-systems based on swarms. In: ACM Symposium on Applied Computer. ACM Press, Orlando, FL, USA (2003) 375 – 379
8. Nagpal, R.: Programmable self-assembly using biologically-inspired multiagent control. In: Proceedings of the 1st Joint Conference on Autonomous Agents and Multi-Agent Systems. ACM Press, Bologna, Italy (2002) 418 – 425
9. Shen, W., Salemi, B., Will, P.: Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots. IEEE Transactions on Robotics and Automation **18** (2002) 1 – 12
10. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Cooperative computing for distributed embedded systems. In: International Conference on Distributed Computing Systems. IEEE CS Press, Wien, Austria (2002)
11. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: ACM Mobicom. ACM Press, Boston, MA, USA (2000)
12. Chen, Y., Schwan, K.: Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In: International Middleware Conference, LNCS 3790, Grenoble, FR (2005)
13. Eugster, P., Guerraoui, R., Handurukande, S., Kouznetsov, P., Kermarrec, A.: Lightweight probabilistic broadcast. ACM Transactions on Computer Systems **21** (2003) 341 – 374
14. Broch, J., Maltz, D., Johnson, D., Hu, Y., Jetcheva, J.: A perfomance comparison of multi-hop wireless ad hoc network routing protocols. In: ACM/IEEE Conference on Mobile Computing and Networking. (ACM Press)
15. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Spatial programming using smart messages: Design and implementation. In: International Conference on Distributed Computing Systems. IEEE CS Press, Tokio, Japan (2004)
16. Nagpal, R., Shrobe, H., Bachrach, J.: Organizing a global coordinate system from local information on an ad hoc sensor network. In: International Workshop on Information Processing in Sensor Networks, Pasadena, CA, USA (2003)